To appear in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000.

Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction

An-Chow Lai and Babak Falsafi
School of Electrical & Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
impetus@ecn.purdue.edu, http://www.ece.purdue.edu/~impetus

Abstract

Communication in cache-coherent distributed shared memory (DSM) often requires invalidating (or writing back) cached copies of a memory block, incurring high overheads. This paper proposes Last-Touch Predictors (LTPs) *that learn and predict the "last touch" to a memory block by one processor before the block is accessed and subsequently invalidated by another. By predicting a last-touch and (self-)invalidating the block in advance, an LTP hides the invalidation time, significantly reducing the coherence overhead. The key behind accurate last-touch prediction is* trace-based correlation, *associating a last-touch with the sequence of instructions (i.e., a trace) touching the block from a coherence miss until the block is invalidated. Correlating instructions enables an LTP to identify a last-touch to a memory block uniquely throughout an application's execution.*

In this paper, we use results from running shared-memory applications on a simulated DSM to evaluate LTPs. The results indicate that: (1) our base case LTP design, maintaining trace signatures on a per-block basis, substantially improves prediction accuracy over previous self-invalidation schemes to an average of 79%; (2) our alternative LTP design, maintaining a global trace signature table, reduces storage overhead but only achieves an average accuracy of 58%; (3) last-touch prediction based on a single instruction only achieves an average accuracy of 41% due to instruction reuse within and across computation; and (4) LTP enables selective, accurate, and timely self-invalidation in DSM, speeding up program execution on average by 11%.

1 Introduction

Distributed shared memory (DSM) is an attractive architecture for building a spectrum of shared-memory multiprocessor servers. DSM servers offer a scalable performance path beyond symmetric multiprocessors (SMPs) [9,4] by maintaining a compatible programming interface and allowing a large number of processors to share a single global address space over physically distributed memory.

ISCA'00, Vancouver, Canada.

Despite software compatibility with SMPs, performance tuning applications on DSMs is often difficult due to the non-uniform nature of memory accesses; remote memory accesses in DSM can take several times longer than local memory accesses. Most DSMs use a cache coherence protocol to allow multiple processors to cache remote data and reduce latency. Communication in DSM often incurs a large coherence overhead because it requires locating the current cached data copies, and either invalidating or fetching the newly produced copy.

One approach to reducing coherence overhead in DSM is to predict when a processor completes accessing a shared block and *speculatively* invalidate the block in advance so that subsequent accesses by other processors find the block available at the directory node. Accurate speculative invalidation can virtually eliminate all invalidation messages and can significantly reduce communication time. In conjunction with sharing [8] or coherence [12] predictors — predicting a subsequent sharer of a memory block — a speculative invalidation can forward a memory block to its subsequent consumer early, potentially eliminating the remote memory access latency in DSM.

Lebeck and Wood [10] proposed the first speculative invalidation technique, called Dynamic Self-Invalidation (DSI). DSI is a heuristic-based technique in which blocks *self-invalidate* themselves upon exiting an application's critical section. As proposed, DSI suffers from four key shortcomings. First, it requires using the coherence protocol to identify candidate blocks for self-invalidation. Memory blocks, however, often exhibit different sharing patterns across application phases. Without either complex adaptive protocols or sophisticated sharing predictors [8], DSI may either reduce opportunity for self-invalidation or result in frequent premature self-invalidations. Second, requests for shared blocks often arrive immediately after a processor exits a critical section, offsetting the gains from self-invalidation using DSI. Third, triggering self-invalidation for all candidate blocks simultaneously creates a burst of messages into the network and at the directory resulting in prohibitive amounts of contention and queueing in the system. Finally, it is non-transparent and assumes that critical section boundaries are exposed to the DSM hardware.

This paper proposes a novel mechanism, *Last-Touch Predictor (LTP)* — loosely derived from Nair's two-level path-based branch predictors [14] — to allow blocks to self-invalidate selectively, accurately, and timely. An LTP predicts the "last touch" to a memory block by one processor before the block is accessed and subsequently invalidated by another. The key intuition behind an LTP is that memory sharing and consequently memory invalidation are triggered by program instructions. Because program behavior is repetitive — e.g., a critical section uses a fixed set of instructions to read and modify data — self-invalidation can be associated with and triggered by program instructions. As compared to DSI, spec-

ulative self-invalidation using LTP: (1) increases the opportunity to reduce invalidation overhead, (2) decreases the frequency of premature invalidation, (3) reduces contention due to self-invalidation traffic, and (4) obviates the need to modify the coherence protocol.

We propose *trace-based correlation* as a fundamental technique to enable accurate last-touch prediction. Much as path-based correlation [14] uses a history of target addresses to predict a branch outcome, trace-based correlation uses a sequence of memory instructions (i.e., an instruction trace) touching a block to predict a last-touch. Instruction reuse within a given computation or across application phases prevents a predictor from associating last-touch prediction with individual instructions. Instead, by maintaining an instruction trace from a coherence miss until a last-touch to a block before an invalidation, a trace-based LTP can uniquely identify and distinguish a last-touch to a memory block both within and across application sharing phases.

We present results from running shared-memory applications on a simulated 32-node DSM to show:

- Our base LTP design, maintaining trace signatures on a per-block basis substantially improves prediction accuracy over DSI to an average of 79% and at best 98% while requiring only 7 bytes of storage per block. In contrast, DSI only achieves an average accuracy of 47%;
- Our alternative LTP design, using a global trace signature table reduces storage overhead to 6 bytes per block but only achieves an average accuracy of 58%;
- Last-touch prediction based on a single instruction (rather than a trace) only achieves an average accuracy of 41% due to instruction reuse within and across computation;
- LTP enables selective, accurate, and timely self-invalidation of blocks in DSM. Speculative self-invalidation using an LTP speeds up program execution on average by 11% and at best by 30% and only increases execution time in one application (by <1%). In contrast brute-force self-invalidation using DSI actually increases execution time in four out of nine applications and achieves an average overall speedup of only 3%.

In the following section, we describe coherence activity in DSM and self-invalidation in DSI. In Section 3, we present the motivation, design, and implementation issues for LTPs. Section 4 describes the mechanisms required to self-invalidate using LTPs. Section 5 presents the simulation methodology and results from running shared-memory applications. Finally, Section 6 presents a summary and concludes the paper.

2 Self-Invalidation Background

Most DSMs use a directory-based coherence protocol to implement a global shared address space over physically distributed memory. In directory-based systems, a directory on every node maintains sharing information for the (physical) memory pages (also referred to as "home" pages) on that node. For every fine-grain (e.g., 32-128 byte) memory block on a home page, the directory maintains a block sharing state and a list of processor identifiers sharing the block. A coherence protocol coordinates sharing of memory blocks among the processors. In this paper, we focus on simple full-map directory write-invalidate hardware coherence protocols such as those implemented in Sun WildFire [4], SGI Origin 2000 [9], and Fujitsu Synfinity [18]. The ideas in this paper, however, are equally applicable to software protocol implementations.

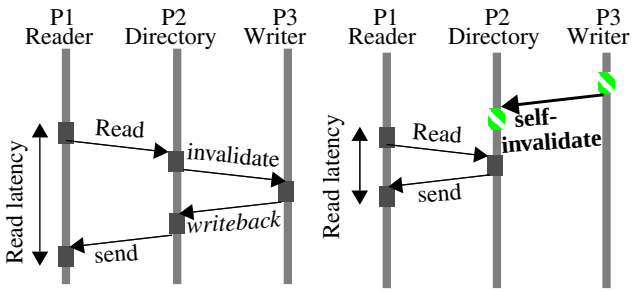


Figure 1: Example protocol operations on a remote read in a conventional DSM (left) and a DSM with self-invalidation (right).

In a write-invalidate protocol, a block can be in one of three protocol states: (1) in the *Idle* state, the block resides at home and is accessed only by the home processor(s), (2) in the *Shared* state, read-only block copies are cached by one or more remote processors, and (3) in the *Exclusive* state, a writable copy is cached by a single remote processor. Upon a read or write request from a remote processor, the block can be forwarded directly when in the *Idle* state. Similarly, upon a read request for a block in the *Shared* state, the directory can directly reply with a read-only block copy. A write to a block in the *Shared* state, however, requires invalidating the read-only copies to grant exclusive access to the writer. Similarly, a read to a block in the *Exclusive* state requires writing back (and optionally invalidating) the writer's copy.

DSM protocols differ in whether, upon a read request, to downgrade a writer's copy and allow the writer to maintain a read-only copy (favoring producer-consumer sharing) [9] or to invalidate the writer's copy (favoring migratory sharing) [16]. Self-invalidation, however, is equally applicable to both protocols and performs either a writeback (in the former case) or a writeback and invalidation (in the latter case). In this paper, we only focus on protocols which invalidate writable blocks upon a read.

Figure 1 (left) illustrates an example sequence of coherence actions requiring invalidation in DSM. When P1 requests a read-only copy of a block, the directory first invalidates and requests a writeback from the current writer, and subsequently sends a read-only copy to P1. The entire read transaction includes four network messages and up to four local memory accesses making remote access latencies much higher than local latencies. Assuming that P3 finishes writing the block early, it could (self-)invalidate the block so that upon the read request's arrival, the block is in the *Idle* state and can be quickly forwarded to P1. Figure 1 (right) illustrates how P3 eliminates the need for invalidation and reduces P1's remote read latency by self-invalidating the block before the arrival of P1's read request.

Self-invalidation can be used in conjunction with other coherence overhead reduction techniques. For instance, together with relaxed memory models, self-invalidation can reduce read latency while the relaxed models allow overlapping the write latency. Similarly, self-invalidation can trigger sharing prediction and speculation in DSMs with adaptive coherence protocols (such as SGI Origin's migratory sharing protocol [9] and SCI's producer-consumer sharing protocol [6]) and DSMs with pattern-based [8] and instruction-based [7] sharing predictors. In the limit, self-invalidation together with accurate sharing prediction can help eliminate remote access latency by always forwarding a memory block to a subsequent sharer prior to an access.

To reduce the remote access latency effectively, the self-invalidation mechanisms in a DSM must accurately identify both "which" memory blocks to self-invalidate and "when" to do so. Remote

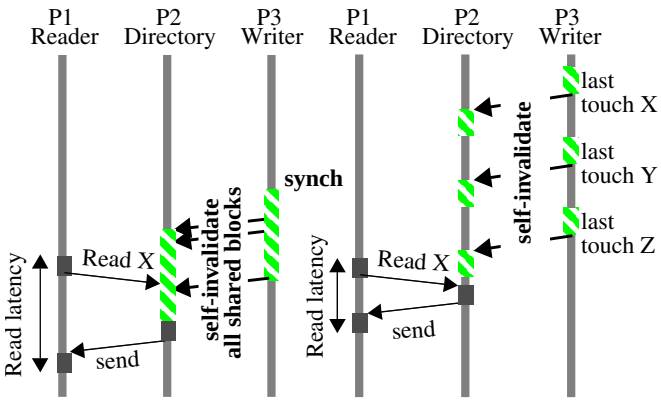


Figure 2: Self-invalidation at synchronization boundaries (left) and using a last-touch predictor (right).

access latencies in DSM are up to tens of times longer than local memory accesses, and as such inaccurate self-invalidation may result in prohibitively large overheads. For instance, self-invalidating the wrong memory blocks or premature self-invalidation can turn processor cache hits into remote memory accesses taking thousands of processor cycles. Similarly, the effectiveness of self-invalidation depends on the fraction of invalidation messages actually eliminated. The latter depends on the number of correctly-identified self-invalidations that actually reach the directory prior to subsequent requests for the self-invalidated blocks.

There are a myriad of proposals for self-invalidation. Software-driven approaches typically use either the compiler [1], the programmer [5], or the binary rewriter (through profiling) [2] to identify opportunities for self-invalidation and insert self-invalidation directives in the code. To self-invalidate blocks accurately, however, software-driven approaches require either complex compiler algorithms or careful annotation by the programmer. Moreover, these techniques require hardware support for directives, are non-transparent, and limit portability. Finally, software-driven techniques use statically inserted self-invalidation directives that are always executed. As such, these techniques can not prevent inaccurate self-invalidations at runtime. In this paper, we focus on hardware techniques for self-invalidation.

2.1 Dynamic Self-Invalidation (DSI)

Lebeck and Wood [10] proposed Dynamic Self-Invalidation (DSI). To identify candidates for self-invalidation, they proposed adaptive protocol schemes which select blocks for self-invalidation only if they are actively shared, i.e., read and written by different processors. Their best scheme is based on "versioning" and maintains write-version numbers at the directory with all the cached copies. Subsequent writes to a block increment the version number at the directory. Upon a block request, the protocol compares the cacher's version number for the block with the one stored at the directory. If the version numbers are different, the block is actively shared and is therefore selected as a candidate for self-invalidation.

They proposed heuristic-based techniques to predict "when" to self-invalidate a block. Their best heuristic uses synchronization boundaries to trigger block self-invalidation. A list of candidate blocks selected by the versioning protocol self-invalidate upon entering/exiting a critical section. Their scheme, however, is non-transparent and requires all synchronization boundaries (including locks and barriers) to be annotated and exposed to the DSM hardware.

Block sharing patterns often vary across program phases [8]. As such accurately selecting candidates for self-invalidation requires either a complex adaptive coherence protocol or an accurate pattern-based sharing predictor [8]. Versioning only selects a block as a candidate for self-invalidation based on the last observed coherence activity on the block. As such, for blocks with varying sharing patterns throughout the application, versioning will always either result in a premature self-invalidation upon a critical section exit or miss the opportunity for self-invalidation.

Memory sharing often begins when one processor exits the critical section. As such, self-invalidating blocks upon exiting the critical section may be *late* and ineffective if subsequent requests arrive before the self-invalidations. Self-invalidating *all* blocks at synchronization point also generates a large burst of invalidation traffic. Bursty traffic may result in congestion in the memory system and the network and potentially increase the memory access times on the execution's critical path thereby reducing performance and offsetting the gains from self-invalidation.

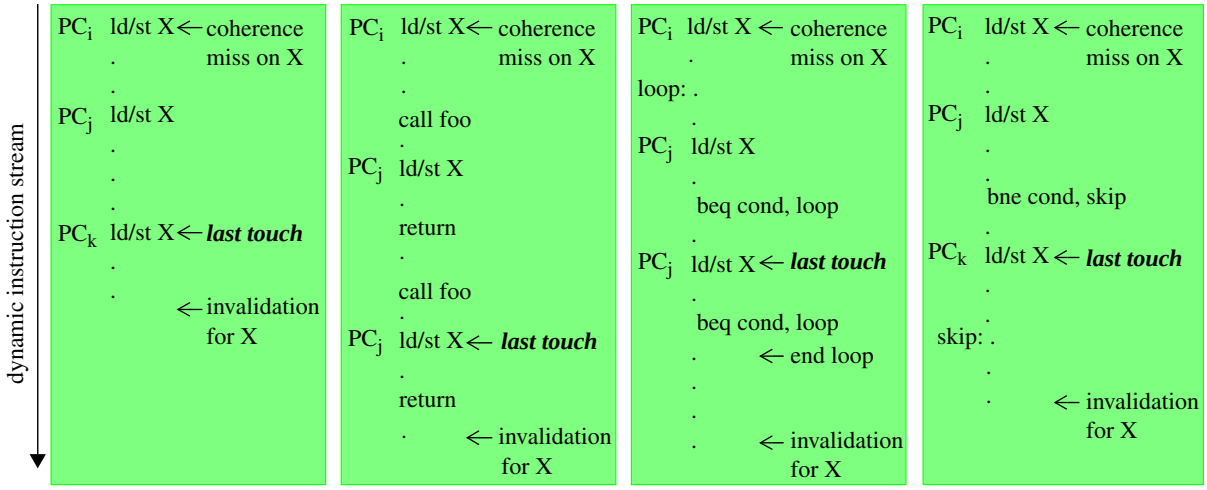
3 Last-Touch Predictors (LTPs)

This paper proposes *Last-Touch Predictors (LTPs)*, mechanisms that accurately predict "when" a block can self-invalidate. The key observation behind an LTP is that memory sharing and consequently memory invalidation are triggered by program instructions. Because the program behavior is repetitive — e.g., a critical section uses a fixed set of instructions to read and modify data — it should be possible to associate self-invalidation with program instructions. Rather than use heuristics to predict when a block should self-invalidate, hardware on every processor can learn and predict the "last touch" by a memory instruction to a shared block before the block is accessed and moved to another processor.

Self-invalidation using an LTP has several advantages. First, a block self-invalidates at the *earliest* possible time — i.e., immediately upon the last reference by the current sharer — maximizing the likelihood that a block is available at the directory prior to a subsequent access. Second, LTP predicts a last-touch and triggers self-invalidation on a per-block basis allowing accurate self-invalidation in the presence of varying sharing activities throughout the application both for a given block and across blocks. For instance, one block may require invalidation immediately after a critical section while other blocks may be referenced across multiple critical sections before requiring invalidation. By predicting a last-touch, an LTP also simplifies DSM protocol design and obviates the need for complex protocol modifications to identify self-invalidation candidates accurately.

Third, by predicting and triggering self-invalidation for individual blocks, LTP helps overlap computation with self-invalidation and reduces the likelihood of memory system and network congestion due to bursty self-invalidation on the execution's critical path. Fourth, LTP is a hardware predictor and as such obviates the need for software annotation which sacrifices transparency. Finally, using LTPs on a per-block basis enables selective self-invalidation, preventing self-invalidation for blocks with low prediction accuracy, and reducing the negative impact of misprediction.

Figure 2 compares self-invalidation at synchronization boundaries using DSI against self-invalidation using a last-touch predictor. The figure (on the left) illustrates that DSI invalidates all actively shared blocks and as such creates congestion both at the cacher (P3) and at the directory (P2). Moreover, because block X self-invalidates late, a subsequent read request for X from P1 must wait until the block arrives. In contrast, LTP (Figure 2 right) selects the



(a) a simple last-touch (b) last-touch in a procedure (c) last-touch in a loop (d) conditional touch

Figure 3: Examples of last-touch to a memory block: (a) a single last-touch in a simple streamlined code, and examples of last-touches embedded in (b) procedures, (c) loops, and (d) conditions.

appropriate blocks for self-invalidation and triggers self-invalidation early.

3.1 Why Use Trace-Based Correlation?

To enable accurate self-invalidation, an LTP must identify the last touch to a block in a given sharing phase and predict it upon a subsequent visit to that phase. Figure 3(a) illustrates the instructions touching a block from the time it is fetched until it is invalidated. A simple approach to identify the last touch to a block is to remember the program counter (PC) of the last instruction touching the block before it is invalidated. By keeping track of all last PCs prior to all block invalidations, a predictor can identify the last-touch to every block in every sharing phase. Triggering self-invalidation would simply require comparing the PC of every instruction touching the block to the list to identify a last-touch.

Unfortunately, common control flow constructs in high-level languages — such as procedure calls, loops, and conditional statements — prevent an LTP from using a single PC to identify a last-touch accurately. For instance, to maintain code modularity, applications often encapsulate common computation into procedures. A procedure may be called multiple times *within* a given sharing phase. The last reference to a block before an invalidation may only happen in the last invocation of the procedure preventing a single PC from accurately identifying a last-touch. In the example in Figure 3(b), PC_j is the last-touch to block X only in the last invocation of the function foo. Procedures may also be called at arbitrary points *across* sharing phases. While a single PC in a given procedure may accurately identify a last-touch in one sharing phase, the PC may not be the last-touch to the same block in another sharing phase, requiring a more accurate mechanism to distinguish last-touches throughout the application's execution.

Similarly, loops complicate last-touch prediction using a single PC. To exploit spatial locality and reduce communication frequency, applications often pack multiple data (e.g., consecutive array elements, or adjacent tree nodes) used in a given sharing phase in a single cache block. Many types of common iterative computation (e.g., stencil, reduction, or computation over graphs) also result in multiple accesses to the same shared datum in the same sharing phase by a single instruction. In the presence of pointers and absence of accurate address-aliasing analysis, a compiler may be unable to hoist a constant reference to single memory block outside the loop, resulting multiple references to a block by a single PC. In the example in Figure 3(c), the instruction at PC_j touches the block twice before an invalidation message arrives for the block sometime after the loop. Because the instruction at PC_j accesses the block twice, it is not possible to identify the last touch using the PC.

Finally, an instruction touching a block before it is invalidated may be embedded in a conditional statement. While conditional branches are often data-dependent, a branch instruction often behaves regularly and in a predictable manner depending on the context in which it is executing. For instance, a branch (in a procedure's body) may be always taken in one sharing phase and not taken in another. Figure 3(d) depicts an example of an instruction (at PC_k) embedded in a conditional. In this example, the branch is not taken and as such the instruction at PC_k is the last touch to the block before it is invalidated. However, if the conditional branch were taken, the instruction at PC_k would not be executed and the last-touch would be by the instruction at PC_j.

To address these problems, we propose a fundamental technique, *trace-based correlation*, to enable accurate last-touch prediction. Much as path-based predictors [14] predict conditional branches dynamically based on correlating a sequence of basic-block addresses, trace-based predictors predict an event (e.g., a last-touch) dynamically based on correlating a sequence (i.e., a *trace*) of instructions. The key intuition behind traced-based correlation is that a trace can be uniquely identified given its instruction sequence and can be distinguished from others. If a trace is repeatable and always leads to (and can be associated with) the same event, a predictor can dynamically learn the trace and accurately predict the event.

In this paper, we define a trace to be a sequence of instructions accessing a block beginning from the first instruction taking a coherence miss until the block is invalidated. In Figure 3, a trace would consist of the sequence $\{PC_i, PC_j, PC_k\}$ for examples (a) and (d) and would consist of the sequence $\{PC_i, PC_j, PC_j\}$ for examples (b) and (c). Assuming that such sequences repeat — e.g., there are always two references to block X (e.g., two array elements stored in a cache block) in the loop — a trace-based predictor would accurately learn and predict the last touch to block X.

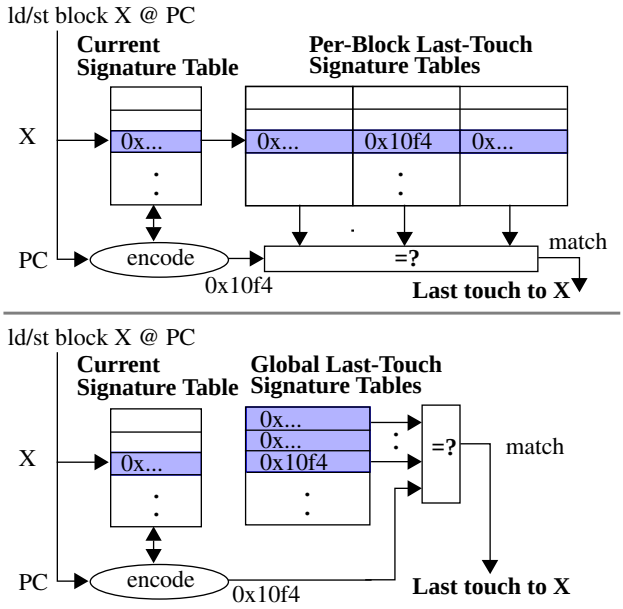


Figure 4: Two-Level trace-based LTPs: a *PAp*-like organization with per-block last-touch tables (top), and a *PAg*-like organization with a global last-touch table (bottom).

Much like path-based correlation, trace-based correlation may result in an inaccurate prediction due to *subtrace aliasing*, the inability to distinguish two traces with identical subsequences. In trace-based correlation, however, subtrace aliasing only arises if the following two conditions are satisfied: (1) one trace is a complete subsequence of another, i.e., the sequence of instructions in the former appears identically and in the same order in the latter, and (2) both traces start from the same PC. For instance, in the example in Figure 3(d), assume that the code starting at PC_i executes in alternating phases in which the conditional branch is taken every other time (e.g., characteristic of red/black SOR computations). The resulting last-touch traces will be $\{PC_i, PC_j\}$ and $\{PC_i, PC_j, PC_k\}$ one of which is a subtrace of the other. As such, trace-based correlation will result in a last-touch misprediction in every invocation of such code.

3.2 Two-Level Trace-Based LTPs

Figure 4 illustrates the anatomy of the two-level trace-based LTPs we propose and evaluate in this paper. To perform self-invalidation, each node of a DSM will include an LTP. The predictor maintains all the memory traces generated by a processor for actively shared blocks — i.e., blocks that are invalidated eventually after they are fetched. Because storing entire traces is prohibitively expensive both in terms of storage and lookup, the predictor instead maintains a small encoding of a trace called a *signature*.

An LTP can accurately distinguish one trace from another and consequently predict a last-touch, as long as distinct traces are represented as distinct encodings. Ideally, the encoding technique would maintain as much entropy from each trace in a signature as needed to distinguish the traces from each other. While LTPs can use arbitrary encoding functions trading off accuracy, cost, and performance, in this paper we use *truncated addition* as the encoding for traces. Our results (in Section 5.2) indicate that truncated addition randomizes the signature bits and enables encoding large traces

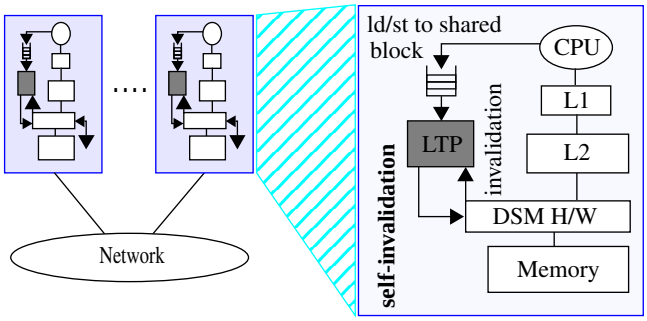


Figure 5: Incorporating an LTP into a DSM.

into a small number of bits that accurately distinguish last-touches from each other.

Much as other two-level branch [14,19] and memory [8] predictors, the first level in an LTP maintains the current history (i.e., signature) of memory accesses while the second table maintains the previously observed patterns (i.e., last-touch signatures). A current signature table at the first level contains a register per block that records an encoding of the trace from the last coherence miss until the last touch to the block. A last-touch signature table at the second level stores a list of previously observed trace signatures.

When learning, an LTP initializes a block's current signature upon a coherence miss with the PC of the faulting instruction. The history table then maintains and updates the current signature upon subsequent memory accesses. When an invalidation message arrives for the block, a trace completes and the corresponding current signature is placed in the last-touch table. Once the last-touch table holds an entry for a block, an LTP begins prediction. Upon a second coherence miss, subsequent memory accesses update the current signature and compare the result against the entries in the last-touch table. A successful match predicts a last-touch by the processor to the block.

We propose two organizations for LTPs, derived from the widely-known two-level branch predictors, *PAp* and *PAg* [19]. Much as in branch predictors, the alternative organizations offer fundamental performance and cost trade-offs for LTPs. Our base predictor design (depicted in Figure 4 top) uses a *PAp*-like organization in which there is a *per-block* last-touch signature table. Our second design (depicted in Figure 4 bottom) uses a *PAg*-like organization and maintains a *global* last-touch signature table for all the shared blocks.

A per-block signature table maximizes the prediction accuracy by eliminating the opportunity for interference among last-touch signatures from multiple blocks. A per-block signature table, however, also increases the storage requirement by maintaining separate last-touch signature tables for every memory block. Moreover, variations in sharing patterns may result in large disparities in the required last-touch table sizes across blocks. For instance, some blocks may only be accessed in a small fraction of the code producing a small number of last-touch signatures, while others may be accessed throughout the code requiring storage for a larger number of last-touch signatures. Large variations in per-block table utilization may result in a prohibitively high storage overhead.

A global table capitalizes on common sharing patterns among blocks and only maintains a single set of last-touch signatures for all the blocks. A global table not only reduces storage for common signatures but also optimizes storage utilization by allowing a variable number of last-touch signatures per block. Unfortunately, a global table also increases the likelihood of subtrace aliasing and interference across blocks, potentially reducing the overall prediction accuracy.

Number of nodes	32
Processor speed	600 MHz
Processor cache	1 MBytes
Memory bus	100 MHz
Local memory/	
Network cache access time	104 cycles
Network latency	80 cycles
Round-trip miss latency	416 cycles
Remote-to-local memory access ratio	~4
Cache block size	32 bytes

Table 1: System configuration parameters.

Benchmarks	Input Data Set	Iter
appbt	12x12x12 cubes	40
barnes	4K particles	21
dsmc	48600 molecules, 9720 cells	400
em3d	76800 nodes, degree 2, 15% remote, distance 2	50
moldyn	2048 particles	60
ocean	128x128	12
raytrace	car	n/a
tomcatv	128x128	50
unstructured	mesh 2K	30

Table 2: Benchmarks and inputs.

3.3 Implementation Issues

To predict a last-touch, an LTP requires access to the memory instruction traces generated by a processor. To learn a last-touch, the system must expose all the invalidation messages for a processor to the corresponding LTP. An LTP requires access to the DSM memory controller to perform a self-invalidation upon a last-touch prediction. Figure 5 depicts how an LTP can be incorporated into a DSM. In the common case, every memory instruction to a shared block will require an access to the LTP. Due to pin bandwidth limitations and to prevent excessive off-chip LTP traffic, the LTP must be implemented on-chip as in other instruction-based predictors [7]. In a highly-integrated DSM design — such as DSMs based on Alpha 21364 [17] — all the DSM hardware is on-chip enabling an easy integration of an LTP.

In conventional board-level DSM designs (e.g., SGI Origin [9]) and DSM clusters (e.g., Sun WildFire [4] and Fujitsu Synfinity [18]) in which the memory controller, the DSM hardware, and possibly a network cache for remote data [3] are implemented off-chip, such a design requires that invalidation messages always be exposed to the processor. Moreover, the DSM hardware must provide an interface for the processor to perform a self-invalidation and flush a shared block (that may reside in an off-chip L2/L3 cache or a network cache) back to the home node.

An LTP can be incorporated into an L2 cache's tag RAM to eliminate LTP's tag overhead. Because self-invalidation is only a coherence optimization and should not interfere with the regular L1 traffic into L2 which may be on the execution's critical path, L2 can buffer the LTP accesses in a queue and only perform them in the absence of traffic from L1. L1 can keep a bit in every block's tag to identify actively shared blocks, i.e., blocks that incur coherence misses, and filter the memory instruction traffic so that only the blocks requiring self-invalidation access the LTP. Alternatively, an LTP can be implemented as a separate direct-mapped or set-associative structure to eliminate the LTP contention in L2.

4 Speculative Self-Invalidation Using LTPs

A key advantage of using hardware-based predictors to trigger self-invalidation is that predictors can monitor and react to their own performance and dynamically select candidates for self-invalidation with a high confidence. Speculation is only beneficial for highly repetitive and predictable invalidations and the corresponding last-touch signatures. Brute-force prediction and speculation may result in frequent premature self-invalidation and incur high overheads. Selective self-invalidation using LTP requires two mechanisms: (1) a mechanism to verify the accuracy of the self-invalidation, and (2) a mechanism to restrict self-invalidation to accurate last-touch signatures. The self-invalidation mechanisms require minimal to no modification to the coherence protocol itself. As such, speculative self-invalidation using LTP can be readily incorporated into a conventional write-invalidate coherence protocol.

To verify speculation, the directory must maintain the identity of the processors self-invalidating their copies. In a read sharing phase, the directory must keep all the self-invalidating readers' identities in a verification mask. In a write sharing phase, the directory must keep the identity of the self-invalidating writer in the mask. Upon a subsequent access, if the block's state changes from read-only to writable or vice versa, all the self-invalidations in the mask are correct and the mask is cleared. If a request arrives from a processor whose bit is set in the verification mask, then the self-invalidation is premature. A verification bit is piggybacked upon a subsequent request to a block and sent to LTPs. To estimate confidence for a predicted signature, we simply associate two-bit saturating counters with each last-touch signature. The two-bit counters are widely used as an effective mechanism to filter low-accuracy predictions.

5 Results

To evaluate LTPs' performance and cost and to gauge the performance improvement using speculative self-invalidation, we run shared-memory applications on a simulated DSM. We use Wisconsin Wind Tunnel II [13] to simulate a 32-node CC-NUMA. Table 1 depicts the configuration parameters for the simulated DSM. We model a board-level DSM implementation using a full-map write-invalidate protocol. To minimize the effect of bursty self-invalidation behavior (characteristic of synchronization-based techniques) we model an aggressive two-stage pipelined protocol engine [15]. To model true communication traffic, we assume a large enough network cache [4] (characteristic of recent DSM designs with aggressive remote caching schemes) to eliminate all capacity/conflict traffic in the CC-NUMA system. We also assume a point-to-point network with a constant latency but model contention at the network interfaces.

Table 2 describes the applications we use in this study (which are similar to those used in [8]) and their input parameters. *Appbt* is a shared-memory implementation of the NAS benchmark. *Barnes*, *ocean*, and *raytrace* are from the SPLASH-2 benchmark suite. *Dsmc*, simulates the gas movements and collisions of a large number of particles in a 3D rectangular box using discrete simulation Monte Carlo method [11]. *Em3d* is a shared-memory implementation of the Split-C benchmark. *Moldyn* is a shared-memory implementation of a CHARMM-like molecular dynamics application. *Tomcatv* is a shared-memory implementation of the SPEC bench-

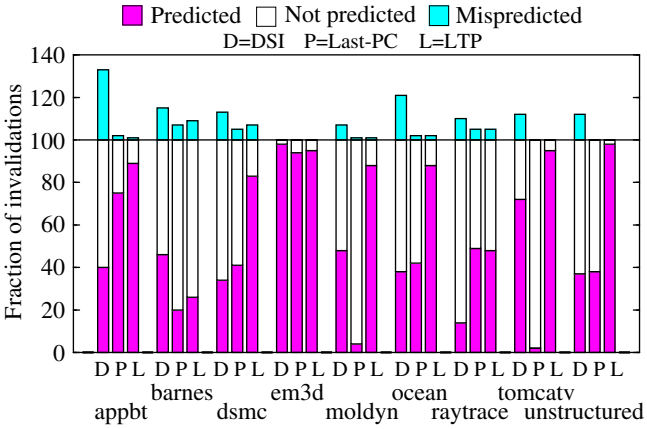


Figure 6: Fraction of invalidations that are accurately predicted, not predicted and mispredicted by DSI, Last-PC, and LTP.

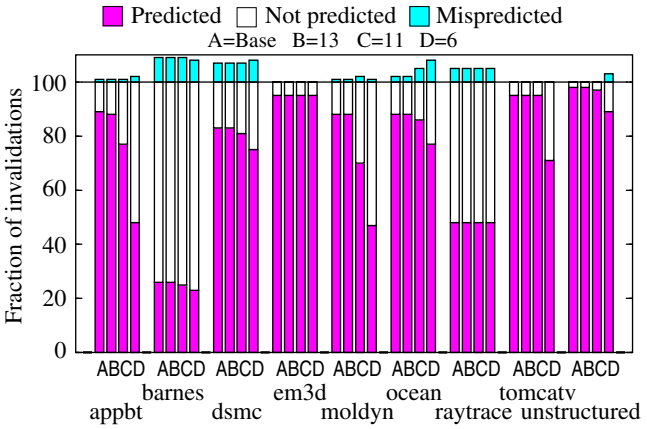


Figure 7: LTP's prediction sensitivity to the signature size.

mark. *Unstructured* is a computational fluid dynamics application that uses an unstructured mesh.

In the following section, we first compare LTP's prediction accuracy against DSI's. In Section 5.2, we investigate the prediction accuracy's sensitivity to signature size. In Section 5.3, we study the accuracy and cost trade-offs of the two LTP organizations we propose. Finally, in Section 5.4 we present execution time results comparing speculative self-invalidation using LTP and DSI. In the rest of this section, we use LTP to refer to a per-block last-touch signature table organization, unless specified otherwise.

5.1 Prediction Accuracy

In this section, we compare LTP's prediction accuracy and coverage against DSI's. To present evidence and motivate the need for trace-based correlation, we also evaluate the prediction accuracy of a simple predictor, *Last-PC*. Last-PC uses the same two-level organization as an LTP but maintains a list of last PCs prior to invalidation rather than a trace signature. Figure 6 compares the performance of the three predictors. The figure plots the fraction of invalidations correctly predicted, not predicted (either due to training or when the two-bit confidence counter is not saturated), and mispredicted (i.e., a premature last-touch prediction).

The graphs corroborate prior evidence that block sharing patterns are often complex and vary across application phases [8]. As a result, DSI's simple versioning protocol fails to detect candidates for self-invalidation accurately. DSI on average only predicts 47% of the invalidations correctly. Moreover, DSI also predicts on average 14% of the invalidations prematurely, which may significantly diminish the gains from the correctly predicted invalidations.

Surprisingly, last-touch prediction using Last-PC performs only slightly worse than DSI and on average predicts 41% of the invalidations correctly. Moreover, using two-bit confidence counters, Last-PC reduces the fraction of mispredicted invalidations to an average of 2%, significantly reducing the potential for premature self-invalidation as compared to DSI. Last-PC's performance, however, varies drastically across applications. While in some applications a single PC can accurately identify a last-touch, in others instruction reuse within and across computation can reduce the fraction of correctly predicted invalidations using a single PC to 2%-3%. In contrast, using trace-based correlation, an LTP achieves the highest prediction accuracy and on average predicts 79% of invalidations correctly and only mispredicts 3% of the invalidations.

Em3d is the most well-behaved application and has the most predictable invalidation patterns. In *em3d*, computation proceeds in a loop and the majority of the blocks are only touched once prior to invalidation. Moreover, the sharing patterns are static and repetitive resulting in a high (> 95%) prediction accuracy in all the predictors.

Tomcatv and *unstructured* also exhibit static sharing patterns. LTP achieves a prediction accuracy of over 95% in these applications. Last-PC fails to predict last-touches because the same instruction references a block multiple times in the same sharing phase. In *unstructured*, the main loop iterates over data values computing a threshold. *Tomcatv*, is a stencil computation in which multiple array elements are stored in the same memory block resulting in multiple references by the same instruction to the block. DSI, however, only achieves an accuracy of 72% and 38% in *tomcatv* and *unstructured* respectively because DSI does not select blocks with migratory sharing patterns (i.e., exclusive block request when the requester has the only read-only copy) as candidates for self-invalidation. Lebeck and Wood [10] found through experimentation that selecting such exclusive blocks as candidates results in frequent premature self-invalidation as a result of subsequent accesses to the block upon exiting a critical section.

In *appbt*, *dsmc*, *moldyn*, and *ocean*, LTP's prediction accuracy is at least as high as 83%. In *appbt*, most last-touches to data blocks are spread among different PCs. The application, however, uses spin-locks in a gaussian elimination phase to synchronize processors. Last-PC predicts most of the data block last-touches, but fails to predict the last-touches to the spin-locks, achieving a prediction accuracy of 75%. Because the spin-locks are not exposed to DSI, it fails to predict a large fraction of the invalidations only predicting 40% of them correctly. Moreover, DSI predicts 25% of the invalidations prematurely.

Moldyn includes a reduction phase in which the same data are read and modified multiple times in a small loop. Multiple references by the same PC in the reduction phase reduce Last-PC's prediction accuracy to less than 3%. Because the reduction phase results in migratory sharing patterns, DSI only predicts 40% of the invalidations correctly. Similarly, in *dsmc* communication occurs through message buffers implemented through a library. Multiple calls to the messaging code in the same computation phase results in multiple accesses to a block by the same instruction preventing Last-PC from accurately predicting invalidations. Subsequent accesses to the main data structure beyond the synchronization in the message buffers significantly reduce DSI's ability to predict and results in a large number of mispredictions.

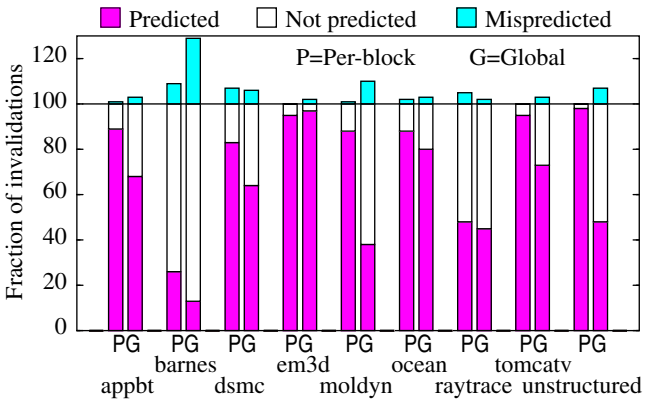


Figure 8: Comparison of prediction accuracy using per-block and global tables.

Ocean implements a red/black SOR algorithm in a computation phase encapsulated in a function invoked twice every iteration. The resulting multiple touches by the function's PCs reduce prediction accuracy in Last-PC to 40%. Sharing blocks in *ocean* often spans beyond critical sections; a block's producer in a critical section reads the block in the subsequent phase. As a result, DSI predicts only 38% of the invalidations accurately and generates 20% of mispredicted invalidations.

In *barnes*, the application's main data structure (i.e., an octree) changes dynamically and frequently. Due to frequent allocation/deallocation of dynamic memory, the last-touch signatures associated with blocks become obsolete reducing the number of correctly predicted invalidations and increasing the number of mispredictions. Moreover, the resulting change in the data structure also changes the traces leading to a last-touch continuously producing new last-touch signatures and decreasing the opportunity for prediction. LTP and Last-PC achieve accuracies of 22% and 20% respectively. Because *barnes* is lock-intensive, DSI manages to predict invalidations after a critical section achieving an accuracy of 42%.

In *raytrace*, there is a global workpool holding the jobs that all processors work on. The workpool is protected by a lock. Invalidations of the global workpool are on the execution's critical path and occur when the lock is accessed, i.e., when jobs are allocated to processors. Because jobs are assigned to one processor at a given time, memory blocks exhibit a migratory sharing pattern and as such DSI exhibits a low prediction accuracy. Both Last-PC and LTP successfully predict the migratory blocks, achieving an accuracy of 50%.

5.2 Sensitivity to Signature Size

A last-touch signature size affects both the predictor's accuracy and implementation cost; a smaller number of signature bits reduces storage cost but increases potential for subtrace aliasing. The ultimate goal is to keep as few signature bits as possible without compromising the prediction accuracy. In this section, we investigate the minimum size of the signature by varying the number of bits in LTP's signature encoding.

The minimum required signature size depends on the encoding technique used; a sophisticated compression algorithm can maximize the entropy in a signature while minimizing the encoding size. The required number of signature bits also depends on the number of distinct trace signatures generated in an application's core of computation. The latter depends on the application's control flow regularity (e.g., the predictability of conditional branches) and instruction footprint size (e.g., the number of different code phases).

	Per-Block		Global	
	ent	ovh	ent	ovh
appbt	2.6	6	1.6	10
barnes	5.2	11	1.6	10
dsmc	7.8	16	1.6	10
em3d	1.0	4	~0	4
moldyn	1.8	5	0.4	5
ocean	2.2	6	0.6	6
raytrace	1.3	4	0.2	4
tomcatv	1.6	5	0.2	5
unstructured	1.5	4	0.2	4

Table 3: Number of signature entries (ent) and overhead in bytes (ovh) for per-block and global tables.

Figure 7 illustrates the prediction accuracies achieved with varying signature sizes. We vary the signature size from 30 bits (minimum number of bits to identify a single PC) to 6 bits. Our results indicate that using truncated addition for encoding, a minimum of 13 bits are required to maintain a high prediction accuracy (as compared to 30 bits) across all benchmarks.

In *em3d*, the computation primarily consists of a tight loop with a single touch between two invalidations to each block. As such, LTP's prediction accuracy is not sensitive to signature size. In *barnes* and *raytrace*, most of the traces are either simple or short, therefore they are not sensitive to signature size. In *appbt*, *dsmc*, *ocean*, and *unstructured* the main computation iterates over several code phases encapsulated in procedures and exhibit large instruction footprints. LTP's prediction accuracy in these applications significantly drops with a small (~6 bits) signature size. A small signature size in these applications prevents LTP from distinguishing traces from distinct application phases. *Moldyn* and *tomcatv* exhibit a small number of distinct signatures. The signatures primarily encode the number of accesses to a block prior to invalidation. A small signature size results in subtrace aliasing in the middle of a sharing phase, reducing prediction accuracy in LTP.

5.3 Implementation Cost

In this section, we evaluate LTP's implementation cost for per-block and global last-touch signature tables. A global table helps reduce storage cost by allowing blocks to share a common storage, but may reduce prediction accuracy due to subtrace aliasing. Figure 8 presents the prediction accuracies using per-block table and global tables. The figure depicts prediction accuracies for a per-block table using 13-bit signatures, and a global table using 30-bit signatures, the minimum signature size necessary to achieve the best prediction accuracy for global tables. Our results indicate that *even* with a much larger number of signature bits, using a global table significantly reduces LTP's prediction accuracy due to subtrace aliasing across blocks; a complete trace for one block is a subtrace for another resulting in inaccurate predictions. A global table on average reduces LTP's prediction accuracy from 79% to 58%. Moreover, a global table increases the fraction of mispredicted invalidations to up to 30%. Our experiments also indicate a very large sensitivity to signature size when using a global table.

There are many scenarios in applications resulting in subtrace aliasing across blocks. For instance in stencil computations (e.g., *tomcatv*) each neighbor reads two of each of left and right neighbors' bordering columns. The computation requires reading the

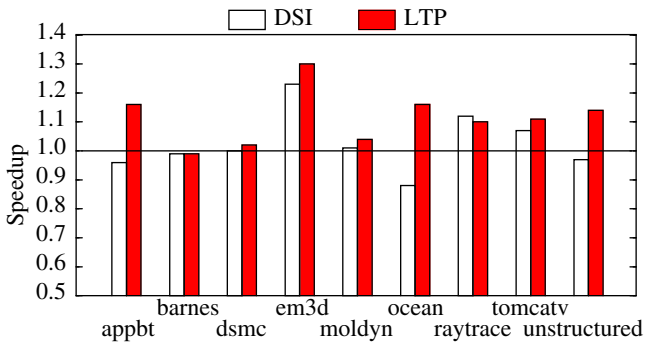


Figure 9: Performance comparison of speculative self-invalidation using an LTP against DSI.

outer column only once and inner the column twice resulting in traces for the outer column blocks becoming subtraces for the inner column blocks.

Table 3 illustrates the number of signatures per block for LTPs using per-block and global tables. The first column for each predictor (depicted as ent) presents the average number of last-touch signatures required over all actively shared blocks. The second column (depicted as ovh) presents the actual per-block overhead in bytes. Both organizations assume a current signature per block and a two-bit saturating counter per last-touch signature.

The table indicates that on average a global table successfully reduces the average number of last-touch signatures in a per-block table from 2.8 to 0.8. However, because of the global tables large signature size, the actual overhead in bytes is on average 6 which is only slightly less than a per-block table's overhead of 7 bytes per actively shared block. These results indicate that for a global table to be a viable predictor organization, the predictor will require a more sophisticated encoding technique that both reduces the signature size and eliminates the subtrace aliasing across blocks.

5.4 Execution Time Results

Speculative self-invalidation's performance depends on a predictor's accuracy, on the incurred burstiness due to self-invalidations, and the self-invalidation timeliness. Even with accurate predictions and a low frequency of mispredictions, self-invalidation performs best only when it is timely and does not generate a burst of messages at the directory.

LTP not only achieves a prediction higher accuracy than DSI, it self-invalidates block as early as possible maximizing the probability that invalidations arrive before a subsequent request. Because LTP uses instruction execution to trigger self-invalidations, invalidation messages also are distributed across program execution, reducing the likelihood of burstiness at the directory. Conversely, DSI may not improve performance as much due to a lower prediction accuracy and lack of timeliness. Moreover, self-invalidation using DSI may actually degrade performance due to a high number of mispredictions and bursty traffic.

Figure 9 presents the speedups achieved by DSI and LTP. The figure indicates that DSI actually *increases* execution time in four out of nine applications and only achieves an average speedup of 3% and at best 23%. In contrast, speculative self-invalidation using an LTP achieves an average speedup of 11% and at best 30%. Moreover, self-invalidation using LTP only reduces execution time in one application by less than 1%. Self-invalidation has little impact on two applications; computation in *dsmc* and high read sharing degree in *moldyn* overlap most of the invalidations, diminishing the effect of self-invalidation.

	Base		DSI		LTP	
	queueing (cycles)	service time (cycles)	queueing (cycles)	timeliness(%)	queueing (cycles)	timeliness(%)
appbt	2	83	916	75	7	98
barnes	8	94	1665	71	8	90
dsmc	9	75	113	48	8	93
em3d	1	98	3283	100	1	100
moldyn	6	79	985	100	3	100
ocean	2	85	309	65	3	98
raytrace	3	126	10	49	12	34
tomcatv	2	79	582	100	2	100
unstructured	13	81	758	100	6	100

Table 4: Average queueing and service time at the directory and the fraction of timely self-invalidations in DSI and LTP.

Table 4 compares LTP's burstiness and timeliness against DSI. The table depicts the average queueing delay, i.e., waiting time, and service time (in processor cycles) per directory message and the fraction of correctly predicted self-invalidations arriving timely (i.e., prior to a subsequent access by another processor). As compared to the base DSM, self-invalidation using DSI increases the queueing delay at the directory on average by three orders of magnitude, resulting in a queueing delay that is an order of magnitude larger than a message service time in the base DSM. Similarly, self-invalidations in DSI are not always timely and on average only arrive 79% of the time prior to a subsequent request. In contrast, the impact on queueing delays due to self-invalidations in LTP is minimal. Moreover, self-invalidations using LTP arrive on average over 90% of the time early.

Despite a high prediction accuracy in *em3d*, queueing delays prevent DSI from competing with LTP. The high number of mispredictions and premature self-invalidations due to DSI limits the performance improvement in *tomcatv* and actually slows down the execution in *appbt* and *ocean*. In *unstructured,* bursty self-invalidation traffic due to DSI increases the queueing delays at the directory. The resulting queueing delays offset the gains from self-invalidation, slowing down the execution time. Similarly, long queueing delay in *barnes* offset the gains from self-invalidation. Surprisingly, DSI speeds up *raytrace* by 11% despite of its low prediction accuracy. Most of *raytrace*'s critical path of execution lies on a critical section. DSI successfully self-invalidates many of the critical sections data blocks, incurs minimal queueing, and improves performance.

Unlike DSI, LTP's performance is directly related to its accuracy when invalidations directly increase coherence overhead on the execution's critical path (i.e., in all applications except for *dsmc* and *moldyn*). In *raytrace*, LTP performs slightly worse than DSI; LTP can not correctly self-invalidate the critical section locks because they spin a variable number of times per visit.

6 Conclusions

In this paper, we proposed Last-Touch Predictors (LTPs) to trigger speculative self-invalidations of memory blocks in a DSM. An LTP predicts a "last touch" to a memory block by one processor before the block is accessed and subsequently invalidated by another. By predicting a last-touch and triggering a self-invalidation in advance, an LTP enables a subsequent access to find the memory

block available at the directory node obviating the need for invalidation and significantly reducing remote memory access latency.

This paper also proposed trace-based correlation as a fundamental technique to enable accurate last-touch prediction. Trace-based correlation uses a sequence of memory instructions (i.e., an instruction trace) touching a block to predict a last-touch accurately. Instruction reuse within and across computations prevents a predictor from associating last-touch prediction with individual instructions. Instead, by maintaining an instruction trace from a coherence miss until a last-touch to a block before an invalidation, a trace-based LTP can uniquely identify and distinguish a last-touch to a memory block throughout an application's execution.

Speculative self-invalidation using LTP has several key advantages over the previously-proposed schemes. An LTP (1) triggers self-invalidation early maximizing the opportunity for reducing memory access latency, (2) is a hardware predictor that accurately identifies candidates for self-invalidation within and across sharing phases, and obviates the need for either complex coherence protocol modifications or software annotation, (3) triggers self-invalidation on a per-block basis reducing the likelihood of memory system and network congestion due to bursty self-invalidations, and (4) enables selective self-invalidation, preventing self-invalidation for blocks with low prediction accuracy, and reducing the negative impact of misprediction.

Results from running shared-memory applications on a 32-node DSM indicated that: (1) our base case predictor, maintaining trace signatures on a per-block basis, substantially improves prediction accuracy over previously-proposed DSI scheme to an average of 79% and at best 98% while requiring only 7 bytes of storage per block; (2) our alternative predictor, maintaining a global trace signature table reduces storage overhead to 6 bytes per block but only achieves an average accuracy of 58%; (3) last-touch prediction based on a single instruction only achieves an average accuracy of 41% due to instruction reuse within and across computation; and (4) LTP enables selective, accurate, and timely self-invalidation of blocks in DSM speeding up program execution on average by 11% and at best by 30% and only increasing execution time in one application by less than 1%. In contrast brute-force self-invalidation using DSI actually increases execution time in four out of nine applications and achieves an average speedup of only 3%.

References

[1] H. Cheong and A. V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[2] T. M. Chilimbi and J. R. Larus. Cachier: A tool for automatically inserting CICO annotations. In *Proceedings of the 1994 International Conference on Parallel Processing (Vol. II Software)*, pages II–89–98, Aug. 1994.

[3] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[4] E. Hagersten and M. Koster. WildFire: A scalable path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, Feb. 1999.

[5] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993.

[6] A. Kägi, N. Aboulenein, D. C. Burger, and J. R. Goodman. Techniques for reducing overheads of shared-memory multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

[7] S. Kaxiras and J. R. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 161–170, Feb. 1999.

[8] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[9] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[10] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[11] B. Moon and J. Saltz. Adaptive runtime support for direct simulation monte carlo methods on distributed memory architectures. In *Scalable High Performance Computing Conference (SHPCC '94)*, pages 176–183, May 1994.

[12] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[13] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Fast and portable parallel architecture simulators: Wisconsin Wind Tunnel II. *IEEE Concurrency*, 2000. To appear.

[14] R. Nair. Dynamic path-based branch prediction. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, pages 142–152, December 1996.

[15] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. Design and performance of parallel high-throughput coherence controllers. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[16] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, Apr. 1994.

[17] M. Report. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, 12(14), October 1998.

[18] W.-D. Weber, S. Gold, P. Helland, T. S. T. Wicki, and W. Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[19] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *24th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 24)*, pages 51–61, December 1991.